PGAS with Lightweight Threads and the Barnes-Hut Algorithm

Hoang-Vu Dang and Alex Brooks and Nikoli Dryden
Department of Computer Science
University of Illinois at Urbana-Champaign, IL, USA
{*hdang8, brooks8, dryden2*}*@illinois.edu*

Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign, IL, USA
Mathematics and Computer Science Division
Argonne National Laboratory, IL, USA
snir@mcs.anl.gov

***Abstract*—We describe a novel runtime system that integrates lightweight threads with a partitioned global address space (PGAS) mode of computation and apply it to the Barnes-Hut (BH) algorithm. Our model combines the power of low-latency, zero-copy, one-sided communication via PGAS with the power of fast context-switching and user-managed preemptive lightweight threads into a hybrid interface. We describe the challenges in designing such a runtime system, analyze approaches and trade-offs, and present benchmark results. Our BH application exemplifies the usage of the model and shows how we can obtain a simple, yet efficient and scalable, algorithm. Our implementation improves on a state-of-the-art implementation by up to 13 times. The hybrid model also improves the performance of various multi-threaded micro-benchmarks on a distributed memory cluster.**

***Keywords*-Barnes-Hut, PGAS, Lightweight thread, Qthreads**

I. Introduction

Trends indicate that future supercomputers will run hundreds of physical threads per node. [1], [2] predict $\mathcal{O}(10^6)$ nodes and $\mathcal{O}(10^3)$ cores per node for the road to exascale computing. They also argue that software stacks must evolve in order to break current obstacles of scalability. Both inter- and intra-node parallelism must be exploited effectively for this to happen. Moreover, problems such as load balancing, latency hiding, and managing communication worsen at this scale. In order to implement efficient applications, new techniques are necessary. One popular development is to integrate asynchronous task parallelism through lightweight threads with separate communication libraries. Several libraries have implemented this type of programming model, one being HCMPI [3].

Lightweight thread libraries will be able to take advantage of the high thread count in future machines. However, the interaction between the thread package and the communication library could be problematic: current implementations of MPI have performance problems when a large number of threads communicate simultaneously, even with intelligent Network Interface Controllers (NICs). These issues can be attributed to course-grain locking and polling overhead, as demonstrated through a simple communication test where threads from different MPI processes communicate in pairs simultaneously (i.e. a ping-pong test). See Figure 1 for the results for various MPI implementations, testing from 1 to 8 communicating threads for messages with 1 byte to 2 MB of data.

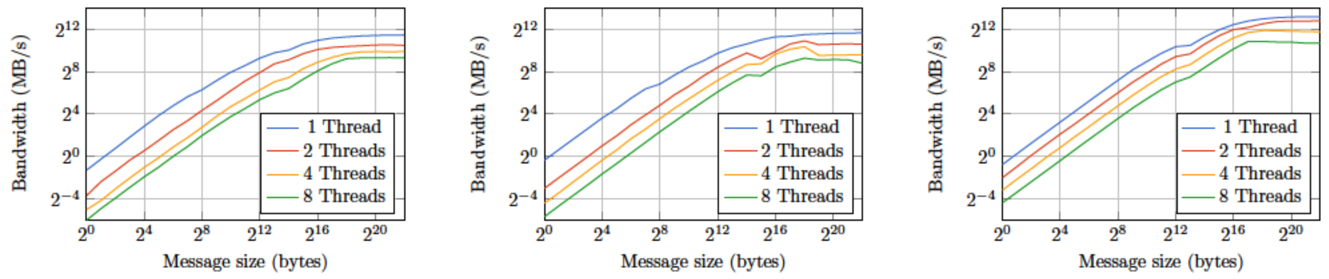
PGAS programming models can offer several advantages over MPI. For instance, the use of global arrays results in programs which are easier to understand. Moreover, PGAS can leverage RDMA technology. However, PGAS languages have not surpassed MPI in popularity due to the engineering efforts required to develop the language and transform existing applications into PGAS-based implementations. There is significant skepticism in the user community about the viability of new languages focused on HPC, hence reluctance to commit to such languages. Libraries are easier to develop and maintain than programming languages and can better interoperate with current code. Further, libraries implemented in high-level languages can leverage the language features so as to appear as language extensions.

We have designed and developed a new C++ runtime library which evolved from our previous work on the Barnes-Hut algorithm using a PGAS model [4], [5]. Our runtime system library provides an intuitive layer of abstraction over PGAS and asynchronous parallelism. The abstraction allows a simple, yet efficient implementation of the Barnes-Hut algorithm on distributed memory, multi-node clusters. Our experiments also show significant improvements for multi-threaded micro benchmarks over GASNet and MPI.

The rest of the paper is organized as follows. Sections II and III describe our PGAS programming model and its current implementation. Section IV describes the n-body simulation problem and the Barnes-Hut algorithm. A state-of-the-art multi-threaded algorithm and implementation in comparison to our method are described in Section V. Section VI discusses experiments and evaluates our implementation of the Barnes-Hut algorithm. Section VII provides some related work. Finally, Section VIII concludes the paper and discusses future work.

II. PGAS and Lightweight thread runtime

We describe in this section the design of PPL, a C++ PGAS parallel libary. PPL is designed for abstracting a model of computation that combines a communication layer which uses one-sided communication, a threading model



(a) MVAPICH on cluster with dual-socket 6-core Intel Xeon processors.

(b) IntelMPI on cluster with dual-socket 8-core Intel Xeon processors.

(c) CrayMPI on a Cray XC30.

Figure 1. Ping-pong results for various MPI implementations on different machines.

which supports task-parallelism, and a memory model which provides an interface for easily manipulating global data structures.

A. Threading Model

We do not distinguish in our design the notion of lightweight and non-lightweight threads. The implementation can freely choose a threading library to realize the idea. However, the full benefit of our design is achieved with lightweight threads that are scheduled by the runtime (i.e. a *task model*).

Our threading interface provides the following methods:

- `spawn`: spawn a thread to execute a function.
- `future`: spawn a thread, but delay execution until a requirement is met.
- `yield`: preempt the executing thread.
- `sync`: synchronization handler that may (temporarily) disable or enable execution of a thread.

B. Communication Layer

The communication layer provides an interface for performing one-sided communication.

The two basic methods are:

- `memget`: perform a read operation from a local or remote memory address into local memory.
- `memput`: perform a write operation from local memory into a local or remote memory address.

When `memget` or `memput` return, they indicate local completion. A handler (`sync`) is associated with each execution of `memget` and `memput` which can be used to poll for remote completion, either by testing or waiting. If an executing thread is required to wait, it will be preempted at least until the communication is finished, providing an opportunity for other *runnable* threads to be executed; an implementation of the communication layer must provide a mechanism for preempting any communicating thread.

C. Memory Model

PPL describes two distinct heaps: a *global heap* and a *local heap*. The local heap contains locations which can be accessed only by locally executing threads, while locations on the global heap can be accessed by any thread. The access to an address on a remote portion of the global heap is done using *global references*.

There are three types of global references described in PPL: global variables (`gvar`), global vectors (`gvec`), and global pointers (`gptr`). A `gvar` is allocated on a single node and provides read-only access to remote nodes.

A `gvec` is similar to a Fortran Co-Array [6]: a distributed matrix such that each process i has local access to a single column i in the `gvec`. The allocation of a global vector is a collective operation. Any implementation must ensure that the local chunks have the same base address on each node, enabling a thread to correctly compute the address of a global vector location from a remote node.

A `gptr` can be used to refer to any location in a local heap or remote portion of the global heap.

Locations in the global heap can be accessed using one-sided operations, such as put or get. These operations may either result in an access to the global heap on the executing node, or to the global heap on a remote node. In the later case, the invocation results in a communication request which will preempt the executing thread.

PGAS languages, such as UPC [7], provide a model of global shared memory, with two restrictions: (1) nodes are single-threaded; and (2) caching is not supported. Work is in progress to add threading to UPC; however, we are not aware of any promising efforts to add caching. PPL escapes both restrictions.

Caching is essential for good shared memory performance. It becomes increasingly important when nodes run a large number of threads, because of the increasing opportunities for *collaborative caching*: a remote location accessed by one thread may also be accessed by other threads on the same node. Many UPC codes perform caching explicitly by copying data from remote nodes to the local heap. PPL provides implicit caching, where remote memory is cached in local memory; however, it does not provide implicit coherence. Implicit coherence does not scale well and most parallel scientific algorithms proceed by well-defined phases; coherence operations can be associated with the beginning or end of a phase.

PPL provides an abstract cache class for which any caching policy may be implemented in software. The caching class provides the following operations: adding an object to the cache, accessing or updating a cache entry, and removing an object from the cache.

In order to provide an opportunity for optimization with regards to accessing remote data, PPL specifies three types of get operations for global pointers: `lget`, `rget`, and `get`. First, `lget` is simply a local get; it is assumed there is a cached copy of the `gptr`. The corresponding cache entry is returned, otherwise an exception is thrown. Next, `rget` ignores any cached copies of the `gptr` and always requests a new copy from the corresponding remote node. This will update the old cached copy and return the new reference once the request is complete. Finally, `get` is a fail-safe generic get operation. If there exists a cached copy of the `gptr`, then the reference is returned. Otherwise, a request for a new copy is sent to the corresponding remote node, updating and returning the new cached copy once the request is complete. Currently there are no changes to the cache when a put operation is performed, for simplicity.

III. PPL IMPLEMENTATION

In this section, we describe implementations of the different PPL modules.

A. Threading Model

In most cases, supporting a large number of threads can be expensive, especially when context-switching is performed regularly. Although the PPL threading model supports any threading library, it is expected that using a lightweight task-based threading model will be more efficient than a standard system-level threading model (i.e. POSIX threads). By using a lightweight task-based threading model, spawning and context-switching latencies are reduced. Further, applications and libraries are unable to explicitly wake system-level threads, since the scheduler is completely dependent on the operating system. Lightweight task-based models typically implement their own scheduler, which includes a load-balancer and mechanisms for preempting and enabling tasks. For these reasons, our first threading model implementation uses a lightweight task-based model, specifically the Qthreads library [8].

Qthreads provides a synchronization primitive which allows tasks to wait on the status of a single bit in memory (i.e. a full/empty bit). This method was originally seen in the Denelcor HEP system for guaranteeing correct ordering of memory operations [9] and is still seen in the Cray XMT architecture as a form of low-overhead synchronization for simple parallel programming models [10]. In order to efficiently couple communication completion with thread scheduling, PPL implements `sync` using the full/empty-bit primitive provided by Qthreads. This ensures that a task waiting for communication completion is preempted and efficiently rescheduled when it can continue execution.

Qthreads provides the ability to adjust the number of workers (threads) and the number of task queues (*shepherds*), as well as provides automatic load-balancing via task stealing between shepherds. In PPL, we fix the number of shepherds and assign one worker per shepherd.

B. Communication Layer

In order to efficiently support a large lightweight thread count with blocking communication calls, it is necessary to offload polling to a separate thread (or threads). In our implementation of the PPL communication layer, we do this by splitting the execution of communication calls between the calling thread and a *communication engine* (CE). The communication engine is defined as a dedicated service thread that executes parts of the communication code which must be executed atomically. This avoids the need for locking and mutual exclusion, which is an important reason why thrashing exists when many threads communicate simultaneously. Further, this design facilitates leveraging more intelligent NICs, since functions can shift between the NIC and CE without affecting other components. We implement the communication layer assuming that a NIC is serviced by a single CE. As modern NICs provide multiple independent virtual interfaces, this assumption is not too restrictive. Larger systems may require multiple CEs, but a suitable partition of traffic can ensure that there is little or no synchronization between multiple CEs.

Communicating threads submit requests to the CE through a *request container* (RC). When a thread performs a blocking communication call (i.e. `memput` or `memget`), the request is submitted to the RC and the thread waits on the associated handler for completion. When a communication request is complete, the CE notifies the scheduler by changing the state of the handler to re-enable the task.

Since communication is completely offloaded to the CE, the scheduler is able to better manage tasks which are waiting for communication requests. Progress on a request continues even if a task is preempted. Therefore, in the general case, it is most efficient to yield a task once it submits a request to the RC, provided that there are other tasks waiting to be scheduled. Likewise, it is ideal to attempt to reschedule a task once notified by the CE that a communication request is complete.

The communication library we use initially for PPL is GASNet [11], in order to simplify development and provide portability.

Through basic testing, we have seen that, compared to the latency of communication with a small amount of data, the overhead of using the full/empty-bit primitive to preempt and re-enable tasks is relative large. In an attempt to hide some of this latency, we use two Pthreads to implement the CE: one thread is dedicated to executing the GASNet

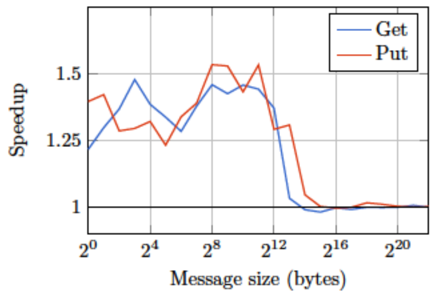


Figure 2. A comparison of put and get latency using one and two threads for the CE.

RDMA calls associated with the communication requests (*Comm Thread*); the second thread is dedicated to managing synchronization for communication requests (*Sync Thread*). When a request is submitted to the RC, it is labeled with the corresponding RDMA operation. Once the Comm Thread of the CE performs the operation, it relabels the request as waiting and places it back in the RC. Periodically, the Comm Thread will test for any completed requests. Once a communcation request is complete, the request is placed in a secondary queue for the Sync Thread to wake the requesting task. This design overlaps the latency of submitting communication to the network (via GASNet calls) with the latency of waking preempted Qthreads tasks.

In order to provide an efficient implementation of the RC, it is necessary to have a thread-safe data-structure which is optimized to allow for multiple producers of data and only a single consumer of data. Thus, we implement the RC as a lock-free MPSC queue [12]. Further, a traditional queue is used to implement the secondary queue since only the two threads of the CE use the queue.

In order to demonstrate the benefits of using two threads for the CE, we ran a simple latency test for put and get operations (see Section VI-C for a more detailed explanation of the test). Figure 2 shows the speedup of using two threads versus one thread in the approach described above. For data sizes up to 8 KB, it improves performance by up to 48% for get and 53% for put. For larger data sizes, the performance is the same due to the synchronization latency being much smaller than the overall communication latency.

The importance of using two threads for implementing the CE does not necessarily come from the improved performance for small messages. For many applications, there may be no benefits. We instead focus our analysis on the benefits with respect to future hardware. Considering an implementation of PPL which places the CE on an intelligent NIC, the latency of notifying the scheduler of a completed communication request should be negligible. By using two threads to implement the CE, we can better understand the behavior of future implementations of PPL which use intelligent NICs. Further, in BH, since there are a large number of relatively small get operations performed, we

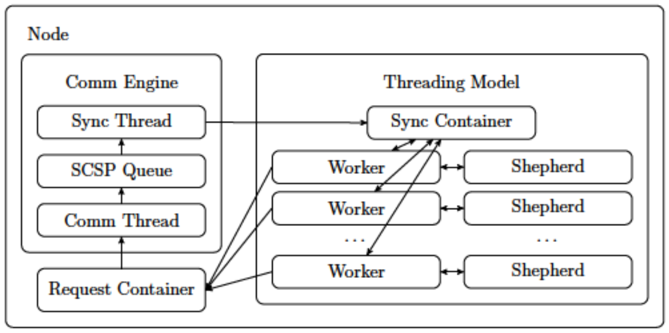


Figure 3. PPL Implementation using Qthreads for the threading model and two threads for the communication engine. *Sync Container* is the Qthreads scheduler.

observe a performance improvement using this approach.

Figure 3 illustrates the interaction between the communication layer and threading model in the current PPL implementation.

C. Memory Model

At the initialization of PPL, a segment of memory is preallocated for one-sided communication. Since the initial implementation of the communication layer uses GASNet, we make use of `gasnet_attach` to perform memory attachment to the network device. The preallocated memory is split into two equal segments for the local and global heaps. `umalloc` [13], a memory manager that dynamically allocates space for objects on custom memory addresses, is used for allocating global references to the custom heaps.

A `gptr` is the basic global reference used for interacting with the memory model. It is a class which contains a single structure holding the remote memory address and the associated node id. Coupled with `gptr` is a software cache engine. The cache is implemented using a concurrent hash table. The key of the hash table is a pair containing a node id and remote address which uniquely identifies a particular `gptr`. Depending on the associated accessor, a query to the cache will check the hash table to determine if a cache entry is already present and perform the appropriate operations. `gvar` and `gvec` are also classes, both implemented using `gptr` as internal data.

IV. Barnes-Hut algorithms

n-body simulations are classical problems of simulating the evolution of a system of n bodies interacting with each other through gravitational and other forces. A direct approach to this problem computes the force of one body with respect to all other bodies, yielding $\Theta(n^2)$ time complexity. This is impractical for a simulation involving a very large number of bodies.

The Barnes-Hut (BH) algorithm is an approximation algorithm to the problem. BH approximates the interaction of a body with a collection of other bodies (cell) deemed *far enough* by considering the collection as a single body, using the center of mass of the collection as its location [14]. The BH algorithm partitions and organizes the 3D space of bodies into a octree. The root of the tree represents a cell that contains all bodies. Each cell is recursively divided into eight child cells that contain the bodies in one octant of the parent cell. The recursion stops when the number of bodies in a cell is below a fixed threshold. Once the tree is built, the center of mass of a group of bodies in a cell can be computed bottom-up. When computing the force of a body, the tree is traversed and the center of mass is used once the cell is deemed far enough. By using this hierarchical partitioning strategy and approximation, BH reduces the complexity of the simulation to $\mathcal{O}(n \log n)$.

Each body can interact with a different number of cells. In order to load balance the computation, the BH algorithm keeps track of the number of forces computed per body at the previous iteration; it uses this count as an estimate of the number of forces to be computed at the current iteration. Bodies are distributed to processors, so that each node has an equal number of forces to compute. The distribution is performed by arranging the bodies on a space filling curve and splitting the curve into equal-weight segments. As a result of this strategy, bodies allocated to a processor are nearby in space, and are likely to need the same cells.

The BH algorithm was implemented in UPC by [4]. In a recent paper, they reimplemented it in C++ using the UPC runtime as the communication layer with several optimizations [5]. The combination of C++ and UPC runtime shows significant speedup over the pure UPC implementation and results in an algorithm that is as good or better than competitors. Thus in this paper we will refer to it as the UPCR-BH and use it to compare to our method.

V. Distributed multi-threading Barnes Hut algorithm

A. UPCR-BH implementation

In UPCR-BH, the octree is built at the beginning of the simulation and then updated after each time-step. The tree is distributed and is stored as a structure linked with global pointers. Each processor holds a copy of a few top levels of the tree.

Each thread is assigned a list of bodies using the scheme described in the previous section. To compute the interactions on a body, a thread traverses the octree starting from the root. We say that a cell is *opened* by a thread if the thread accesses the children of the cell. If a remote cell must be opened, and its children are not local, the worker thread will perform communication in order to *localize* the children. We want to avoid localizing the same cell multiple times by using some caching scheme. [4] describes three different schemes. The common idea of the three schemes is to maintain a number of concurrent hash tables used to indicate if a cell has been previously localized. The difference between the three algorithms is the different communication policy used for each, i.e. which worker performs the communication and how the hash tables are distributed.

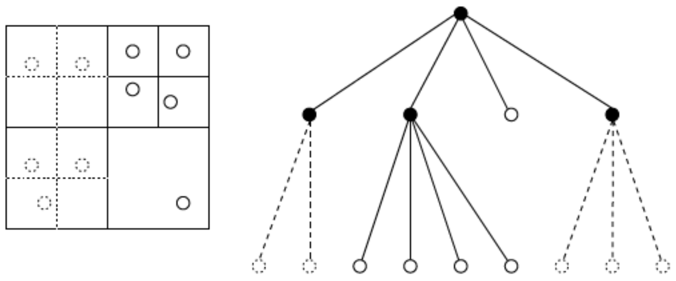


Figure 4. View of a single BH process for a quad-tree (on the right) of a 2D space partition (on the left). Bodies are stored at leaf nodes and are distributed into 2 processes. The dotted lines and circles represent the remote subtrees and subcells.

Among the three algorithms, the *biased and distributed* algorithm performs the best. It dedicates a thread for communication and maintains a hash table for each of the workers and at the communication thread. Since a worker does not know about other worker hash tables, multiple threads can still submit the same cell to the communication worker. However, due to the private hash inside the communication thread, there is no redundant communication throughout the force computation. This implementation is the fastest among the three due to better request management, which reduces pressure on the concurrent request queue of the communication worker. On the other hand, this method uses more memory.

The force computation phase is the most time consuming phase in a BH time step. Its implementation in UPCR-BH is complicated due to the management of many different concurrent data structures. Further, intra-node load balancing between workers uses a simple-guided scheduling approach which may result in superfluous locking. These issues motivated us to develop a more intuitive and efficient design of the force computation phase.

B. New BH Force Computation

The BH implementation can be simplified using the PPL library described in the previous sections: localizing a cell is equivalent to caching it. With lightweight threads, it becomes possible to spawn a thread for each body; the thread scheduler ensures that these threads are scheduled efficiently on existing hardware resources.

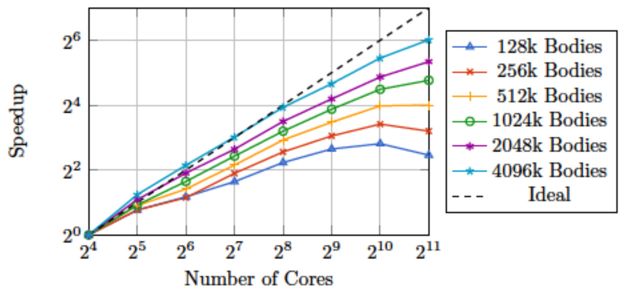
Pseudo-code for the new force computation phase can be seen in Algorithm 1. Function `BH-Force` is the main routine of the algorithm and is executed each step for each compute node. The `ComputeForce` function computes the interaction of a single body on a cell. The `Localize` function is used to localize children of unlocalized cells. Each execution of `ComputeForce` is done by a single

Algorithm 1 BH Force Computation with PPL

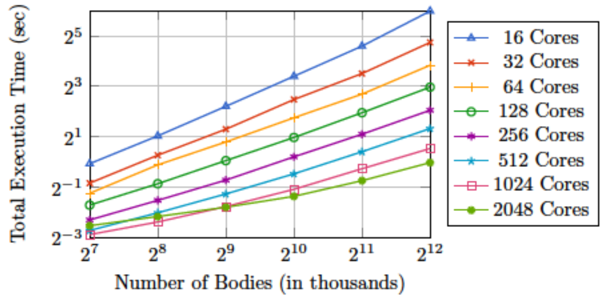
```
Require:
  node.handler : future(Localized(node))
  L: list of local bodies.

  procedure BH-FORCE
    for all body ∈ L do
      spawn(ComputeForce(root, b)).sync()
    end for
  end procedure

  procedure COMPUTEFORCE(node, body)
    if NeedOpen(node) then
      if !node.local then
        node.handler.sync()
      end if
      OpenCell(node, body)
    else
      BodyCellUpdate(node, body)
    end if
  end procedure

  procedure LOCALIZE(node)
    if node.local then return
    end if
    for child ∈ node.childs do
      node.local = &child.gptr.get()
    end for
  end procedure
```

spawned task using `spawn`, whilst `future` is used for spawning cell localization tasks and executing them in the future. `NeedOpen` checks a body against the current cell and decides whether to open it. Depending on the decision, `OpenCell` computes the appropriate calculation on the cell and recursively performs `ComputeForce` on child nodes, while `BodyCellUpdate` updates the body position.

Redundant communication is handled in three ways. First, each node maintains a `future` of each localization task. Once a task performing cell localization is spawned, its handler is synchronized. If a later force computation task attempts to localize the same cell, the `future` handler will return the spawned task handler. Second, inside the `Localize` function, another check of the local pointer is performed at the beginning of the function and will return early if the pointer indicates that the node is already localized. The previous two steps do not eliminate redundancy in race conditions, thus we provide another fail-safe layer. Each remote pointer associated with a child node is implemented using a `gptr`. A single `gptr.get()` will return a cached value if the corresponding `gptr` has been previously requested, otherwise a communication request is submitted to retrieve the data.



(a) PPL BH speedup compared to ideal in strong scaling tests.



(b) PPL BH execution time in strong scaling tests.

Figure 5. PPL BH strong scaling results.

Figure 4 gives an example of a tree structure at the beginning of a force computation step. The dotted line represents the remote view of a process for a particular cell which is implemented using `gptr`.

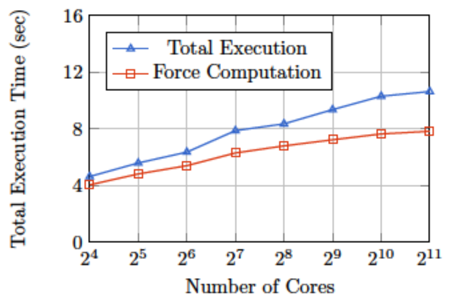
In our algorithm, the threading library preempts (via `yield`) waiting tasks in two cases. First `ComputeForce` tasks can yield after the node handler performs a `sync`. Second, a `Localize` task can yield if `gptr.get()` performs communication (i.e. the `gptr` is not already cached). These tasks will be disabled until the synchronization condition is met; a task is re-enabled after the completion of localization and after the completion of communication, for `ComputeForce` and `Localize`, respectively. We do not implement an explicit load-balancing policy and currently leave all intra-node load-balancing to the Qthreads library.

VI. Evaluation

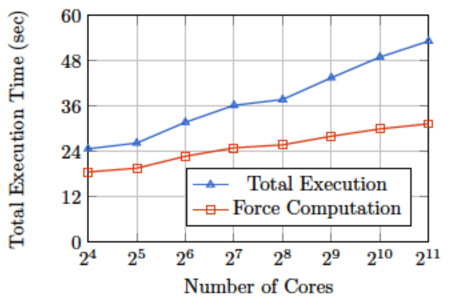
We now discuss our testing methodology and results for our Barnes-Hut implementation, including a comparison with a state-of-the-art BH implementation, UPCR-BH [5]. Further, we present results of a micro-benchmark testing the latency of one-sided communication for GASNet, MPI, and PPL.

A. Test Platform and Methodology

We conducted evaluations on two systems. The first is Taub, a cluster at the University of Illinois at Urbana-Champaign. Each compute node has at least 24 GB of RAM and two Intel HP X5650 six-core processors running at 2.66 GHz. Each core has a private 32 KB L1 data cache and a

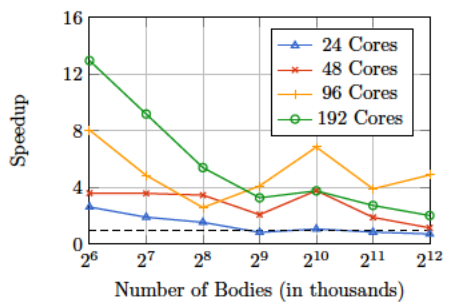


(a) PPL BH weak scaling results with 512k bodies per node.

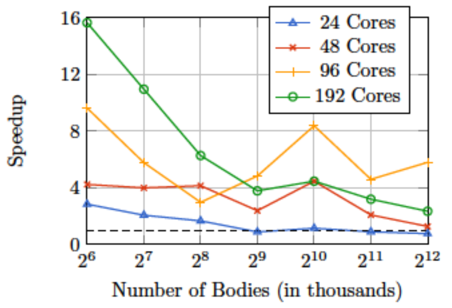


(b) PPL BH weak scaling results with 2048k bodies per node.

Figure 6. PPL BH weak scaling results.



(a) Speedup in total execution time of PPL BH relative to UPCR-BH implementation.



(b) Speedup of the force computation phase of PPL BH relative to UPCR-BH implementation.

Figure 7. Comparison between UPCR-BH and PPL BH.

private 256 KB L2 cache, and share a 12 MB L3 cache. The network uses a QDR InfiniBand interconnect. On Taub, the PPL BH implementation was run with 10 shepherds with one worker thread each and two communication threads; the UPCR-BH implementation was tested using 12 threads.

The second system is the Stampede supercomputer, located in the Texas Advanced Computing Center. It consists of 6400 compute nodes each with 32 GB of memory and two Intel Xeon E5-2680 eight-core processors running at 2.7 GHz. Each core has a private 32 KB L1 data cache and a private 256 KB L2 cache, and share a 20 MB L3 cache. Nodes are interconnected with a 56 GB/s FDR InfiniBand interconnect in a 2-level Clos fat tree topology. Each node also has at least one Intel Xeon Phi SE10P co-processor, but these were not used in our evaluation. On Stampede, the PPL BH implementation was run with 12 shepherds with one worker thread each and two communication threads.

On both systems, the C/C++ compiler was GCC 4.7.1. PPL used GASNet 1.22.4 as its underlying communication library; UPCR-BH used the Berkeley UPC 2.18.0 runtime. Input bodies were generated with the Plummer model [15]. We ran four time steps and excluded the first two, averaging the remainder. All computations involved a time step of 0.025 seconds and a tolerance of 0.5, and were done in double precision.

B. BH Results

We ran two sets of experiments to evaluate our PPL BH implementation: weak and strong scaling tests on the Stampede system and comparison tests with UPCR-BH on the Taub system.

Figure 5a shows the strong scaling results for PPL BH on Stampede, using one node as a baseline. For 4096k bodies, we achieve nearly linear speedup until we reach 2^9 threads, after which we drop slightly below the ideal speedup. However, as can be seen in Figure 5b, we continue to achieve reasonable speedup in total execution time at any combination of bodies and cores. We observe that PPL BH scales better as the body count increases.

Our weak scaling results for 512k bodies per node and 2048k bodies per node are presented in Figures 6a and 6b, respectively. In both cases, we see that PPL BH continues to scale well as the problem size and thread count increase. Computation time grows roughly logarithmically, as expected.

Finally, we compare PPL BH with UPCR-BH in Figure 7a using total execution time. Both implementations were run for problem sizes with total body counts ranging from 64k to 4096k on two to sixteen nodes on the Taub system. We find that PPL BH outperforms UPCR-BH for all body counts once we reach four nodes or more; at sixteen nodes, PPL BH is between 2 and 13 times faster than UPCR-BH, depending upon the problem size.

Our performance increase primarily comes from improvements in the force computation phase. The speedup for this phase compared to UPCR-BH is shown in Figure 7b, and closely matches the total execution time speedup seen above. This is primarily due to the use of lightweight threads to overlap computation more efficiently, and hence make better use of the available processor resources.

C. One-sided Operation Results

To demonstrate the scalability of PPL's design, we conducted micro-benchmark experiments to evaluate the performance of one-sided communication for PPL in comparison with MPI+Pthreads and GASNet+Pthreads. The experiments perform large amounts of remote put and get operations for varying size data transfers and varying number of threads. The experiments were run on the Taub system. Each thread is given an identification and paired with a thread on the other node. Each pair of threads performs puts and gets using separate chunks of the global heap.

MVAPICH2 version 2.0 is used for the MPI+Pthreads experiment and GASNet version 1.22.4 for the GASNet+Pthreads experiment. For the MPI+Pthreads benchmark, an MPI memory window is used for each pair of threads. The synchronization is done on each window after each put and get to ensure remote completion. There is no explicit synchronization for GASNet since we use the blocking put and get syntax. GASNet's specification ensures remote completion when these functions return.

Figures 8a and 8b show the results for put and get using 1 thread per node, respectively. As expected, PPL results are worse than MPI+Pthreads and GASNet+Pthreads due to the extra overhead of managing communication and lightweight tasks. This overhead is merely 2-3 μsec in both cases, thus our implementation of the PPL communication engine has relatively low overhead.

Figures 8c and 8d show the results for put and get using 10 threads per node, respectively. For all data sizes, PPL performs better than GASNet+Pthreads due to better management of simultaneous communication operations. This holds true for MPI+Pthreads up to 16 KB. There are a few reasons which could explain why MPI+Pthreads performs better at larger sizes. First, for communication larger than a certain threshold, many MPI implementations such as MVAPICH split messages into pieces and switch to another mode of communication, resulting in smaller overall latency for transfer setup. This approach proves to scale better and achieve higher performance for large messages on clusters with InfiniBand interconnects [16]. Second, the method MVAPICH uses to poll for communication completion is different from PPL. Specifically, MPI polls the network for any communication completion from nearly any MPI function invocation. This results in identifying completed communication quicker, but at the same time increases overhead of all associated MPI functions. On the other hand, PPL currently only supports polling specific communication requests due to GASNet's specification on network polling. It is expected that the advantage of MVAPICH over PPL for large messages will disappear when we implement PPL over the same communication layer that is used by MVAPICH.

Nevertheless, when the communication mode is the same (i.e. compared to GASNet+Pthreads), PPL consistently performs better with an average speedup of 1.5 times. Up to 16 KB, PPL achieves an average speedup of 2.8 times compared to MPI+Pthreads.

VII. Related Work

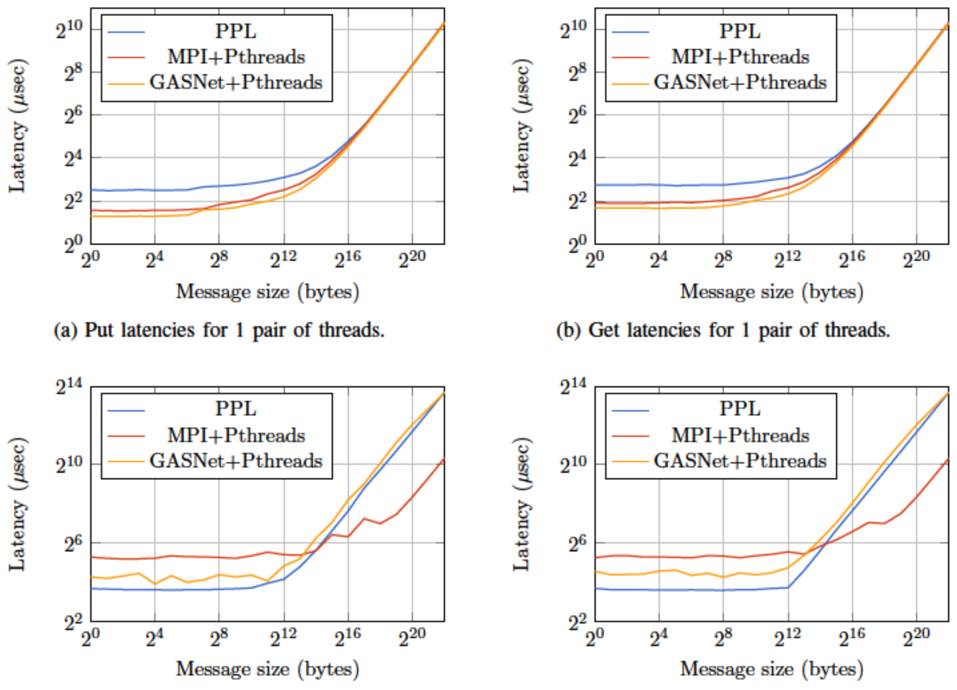
On large scale systems, hybrid programming models have become part of mainstream research in recent years. A very common approach is to incorporate an MPI implementation with a threading library such as OpenMP [17], Habanero-C [3], and SMPSs [18]. MPI implementers are also looking for improving MPI performance by integrating threading libraries with the communication layer [19]. The hybrid MPI+Pthread library approach has shown improvements for several algorithms on multi-core distributed memory clusters. n-body simulations are among those studied, as shown in [20], [21]. These implementations, however, are based on the locally essential tree initially proposed in [14], which does not allow overlapping computation and communication very effectively.

PGAS programming languages such as Chapel [22] and X10 [23] have dedicated threading mechanisms, while UPC [7] and CAF [6] have added multi-threading support through language extensions. Chapel and X10 do not provide lightweight threading support; however, several efforts have been made to add lightweight threading support, such as in [24], [25]. Since lightweight thread features are added later to the design, they are still adhoc and not widely adopted. The PPL PGAS library natively supports lightweight threads and can be easily extended to any threading library through class inheritance.

We are not aware of any PGAS and lightweight thread implementation of the Barnes-Hut algorithm. Several comparisons to different BH implementations with other programming models, such as Charm++ [26] and PEPC [27], have been investigated in [5], [4].

VIII. Conclusion

We have presented the design and an implementation of PPL, a new C++ parallel runtime system which provides an intuitive abstraction over the PGAS model and asynchronous lightweight threads. We have demonstrated how PPL, implemented by integrating a PGAS communication layer with the Qthreads lightweight threading model, can be used to implement a parallel Barnes-Hut implementation easily and efficiently. Further, our BH implementation scales well for large thread counts and significantly out-performs



(a) Put latencies for 1 pair of threads.

(b) Get latencies for 1 pair of threads.

(c) Put latencies for 10 pairs of threads.

(d) Get latencies for 10 pairs of threads.

Figure 8. Put and get latencies for 1 and 10 pairs of threads for data sizes ranging from 1 byte to 4 MB.

the current state-of-the-art multi-threaded BH algorithm. It does so by utilizing lightweight threads to more efficiently interleave communication and computation, making greater use of available computational power. Micro-benchmarks affirm this by showing the consistency and performance of PPL in comparison to GASNet and MPI in multi-threaded settings.

As supercomputers advance toward exascale, future parallel applications will need to make greater use of inter- and intra-node parallelism in the face of increasing communication delays. The use of C++ allows extending and reimplementing features of PPL easily through class inheritance and polymorphism. This makes it an attractive platform for experimenting with different combinations of programming models. We plan to continue to improve PPL, extending it to support different underlying communication and threading mediums, while evaluating its performance on a wide range of problems.

Acknowledgment

The research presented in this paper was funded through the NSF CISE CCF grant 1337217 and the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories, in the context of the Extreme Scale Grand Challenge LDRD project. The work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

References

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, “Exascale computing study: Technology challenges in achieving exascale systems,” *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[2] V. Sarkar, W. Harrod, and A. E. Snavely, “Software challenges in extreme scale systems,” in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012045.

[3] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, “Integrating asynchronous task parallelism with MPI,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 712–725.

[4] J. Zhang, B. Behzad, and M. Snir, “Optimizing the barnes-hut algorithm in UPC,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 75.

[5] ——, “Design of a multithreaded barnes-hut algorithm for multicore clusters,” Tech. Rep. ANL/MCS-P4055-0313, MCS, Argonne National Laboratory, Tech. Rep., 2013.

[6] R. W. Numrich and J. Reid, “Co-Array Fortran for parallel programming,” in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.

[7] UPC Consortium and others, “UPC language specifications v1. 2,” *Lawrence Berkeley National Laboratory*, 2005.

[8] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[9] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *25th Annual Technical Symposium*. International Society for Optics and Photonics, 1982, pp. 241–248.

[10] D. Mizell and K. Maschhoff, "Early experiences with large-scale Cray XMT systems," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–9.

[11] D. Bonachea, "GASNet specification, v1. l," *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.

[12] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 267–275.

[13] A. Righi, "umalloc," http://minirighi.sourceforge.net/html/umalloc_8c.html, accessed: 2014-10-17.

[14] J. K. Salmon, "Parallel hierarchical n-body methods," Ph.D. dissertation, California Institute of Technology, 1991.

[15] S. Aarseth, M. Henon, and R. Wielen, "A comparison of numerical methods for the study of star cluster dynamics," *Astronomy and Astrophysics*, vol. 37, pp. 183–187, 1974.

[16] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen, "Design and implementation of MPICH2 over InfiniBand with RDMA support," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 16.

[17] R. Rabenseifner, G. Hager, G. Jost, and R. Keller, "Hybrid MPI and OpenMP parallel programming," in *PVM/MPI*, 2006, p. 11.

[18] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid MPI/SMPSs approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010, pp. 5–16.

[19] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi," in *Parallel Processing and Applied Mathematics*. Springer, 2014, pp. 559–570.

[20] T. V. T. Duy, K. Yamazaki, K. Ikegami, and S. Oyanagi, "Hybrid MPI-OpenMP paradigm on SMP clusters: MPEG-2 encoder and n-body simulation," *arXiv preprint arXiv:1211.2292*, 2012.

[21] H. Rein and S.-F. Liu, "REBOUND: an open-source multi-purpose n-body code for collisional dynamics," *arXiv preprint arXiv:1110.4876*, 2011.

[22] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[23] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[24] K. B. Wheeler, R. C. Murphy, D. Stark, and B. L. Chamberlain, "The Chapel tasking layer over qthreads," *CUG 2011*, 2011.

[25] J. Paudel, O. Tardieu, and J. N. Amaral, "Hybrid parallel task placement in X10," in *Proceedings of the third ACM SIGPLAN X10 Workshop*. ACM, 2013, pp. 31–38.

[26] L. V. Kale and S. Krishnan, *CHARM++: a portable concurrent object oriented system based on C++*. ACM, 1993, vol. 28, no. 10.

[27] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon, "A massively parallel, multi-disciplinary barnes–hut tree code for extreme-scale n-body simulations," *Computer Physics Communications*, vol. 183, no. 4, pp. 880–889, 2012.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.